Building FIFO and Priority-Queuing Spin Locks from Atomic Swap

Travis S. Craig
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
travis@cs.washington.edu

Technical Report 93-02-02

February 1, 1993

Abstract

We present practical new algorithms for FIFO or priority-ordered spin locks on shared-memory multiprocessors with an atomic swap instruction. Different versions of these queuing spin locks are designed for machines with coherent-cache and NUMA memory models. We include extensions to provide nested lock acquisition, conditional locking, timeout of lock requests, and preemption of waiters. These locks apply to both real-time and non-real-time parallel systems and we include a comparison of the traits of several lock schemes aimed at those environments. Our main technical contributions are our techniques and algorithms that provide tight control over lock grant order, use only the atomic swap instruction, use at most one (local only) spin for lock acquisition and no spinning for lock release, and need only $O(L + P)$ space on either a coherent-cache or NUMA machine.

This research was supported in part by the National Science Foundation under grant number CCR-9200858 and by a National Defense Science and Engineering Graduate Fellowship.

1 Introduction

To maintain the logical consistency of shared data structures is a common problem on shared-memory multiprocessors. One standard technique for maintaining consistency is to protect each structure or set of structures with a lock. A process that needs to access a structure must request the lock for that structure. After acquiring the lock, that process is guaranteed exclusive access to the structure until it releases the lock. Other processes that request the lock in the meantime must wait. When a process releases the lock, one of the waiters acquires the lock and may proceed to use the structure while the others continue to wait.

Processes wait for locks either passively ("relinquishing" the CPU) or actively ("spinning"). To wait passively, a process registers its request for the lock and blocks in order to allow other processes to use its processor while it waits. When the lock is released, one of the registered waiters is chosen to acquire it. The chosen waiter is unblocked and runs when scheduled. To wait actively, a process typically enters a tight loop in which it repeatedly checks the status of the lock and/or attempts to acquire it. Once it acquires the lock, it simply proceeds to access the protected structure.

Both Anderson [And90] and Mellor-Crummey and Scott [MCS91] provide extensive discussions of the relative strengths and weaknesses of relinquishing and spinning. Together with Graunke and Thakkar [GT90], they discuss numerous architectural and software considerations and evaluate several spin lock schemes in the context of non-real-time parallel systems. Molesky, etal [MSZ90] and Markatos and LeBlanc [ML91] discuss the considerations for real-time systems. Each of these sources also introduces one or more schemes for *queuing spin locks*, in which each waiting process spins on a separate location (a separate cache block on coherent-cache machines) and ownership of the lock is explicitly passed from process to process.

In non-real-time systems, where a major goal is to reduce average-case execution times, the main potential benefit of a queuing spin lock is that the waiting processes don't all spin on one location. In fact, each process can spin on a location that is in some way local to its processor, thereby reducing the load on the interconnect between processors and memory. This trait is particularly important during a period of high contention, when there are several waiters for one lock. In addition, a queuing spin lock might grant requests in FIFO order, providing a sort of "fairness" to the processes and guaranteeing against starvation. We provide a FIFO queuing spin lock for coherent-cache machines and one for non-uniform memory access (NUMA) machines.

The ability to control the order of granting requests is the main appeal of queuing spin locks in real-time systems, where a major goal is to reduce worst-case execution times. One way to help bound execution times is to bound the waiting time for a lock by using a FIFO granting order. In priority-scheduled real-time systems, however, a common goal is to reduce priority inversion (where a lower-priority process is holding a resource that is needed by a

higher-priority process [SRL90]). Therefore, we are interested in mechanisms to grant locks to waiting processes in priority order. We provide a priority-queuing spin lock for coherent-cache machines and one for NUMA machines.

Beyond the factors already mentioned, we seek to broaden the practical applicability of our locking schemes by extending them with the following features:

- Nesting — Allow a process to hold more than one lock at a time.
- Timeout — Allow a process that's waiting for a lock to rescind its request if it hasn't been granted by some time limit.
- Conditional Request — Allow a process to request a lock on the condition that the lock is available. If the lock is being held by another process, then the requester returns immediately with an indication that it did not receive the lock.
- Preemption of Waiters — Ensure that a lock is not granted to a waiter that is not running, but don't prevent the scheduler from preempting a process that is waiting for a lock.

1.1 Previous Work on Queuing Spin Locks

The works cited earlier [And90, GT90, MCS91] use queue locks for general-purpose (non-real-time) multiprocessor synchronization in coherent-cache and/or NUMA machines. Their goal, therefore, is good average-case behavior.

Molesky, Shen, and Zlokapa [MSZ90] start from work by Burns [Bur78] and their own work on characterizing round-robin buses to produce schemes that grant locks in round-robin fashion, using test-and-set as their only atomic instruction. By doing so, they at least bound priority inversion by bounding the waiting time for any given processor to get a lock. They also cover some subtle hardware considerations for attaining predictability.

Markatos and LeBlanc [ML91] attack the same priority spin lock problem as we do, but for (1) coherent-cache machines with test-and-set instructions and (2) coherent-cache and NUMA machines with both swap and compare-and-swap instructions. They base their solutions on their improved version of Burns [Bur78] and on Mellor-Crummey and Scott [MCS91], respectively. In the course of their work, they develop a number of important concepts that are basic to priority spin locks. We cite their work when we present or use their concepts. Our work differs from theirs in that we solve the problem for coherent-cache and NUMA machines with atomic swap instructions. We offer a comparison of our schemes in Section 6. Another difference between our work and theirs is in the extended features that we present in Section 5.

1.2 Organization

The rest of this paper is organized as follows.

- Section 2 contains the definitions and architectural model that we use.
- Section 3 presents our basic FIFO and priority-queuing spin lock schemes for a machine with coherent caches.
- Section 4 presents our basic schemes for a machine with non-uniform memory access and no coherent caches.
- Section 5 describes extensions to our basic schemes to improve the priority queue and to provide nesting, timeout on acquisition, conditional acquisition, and preemption of waiters.
- Section 6 discusses the state of our work and compares our results with other queuing spin lock schemes.
- Section 7 summarizes our technical contributions to date and lists work remaining to be done.

2 Definitions and Model

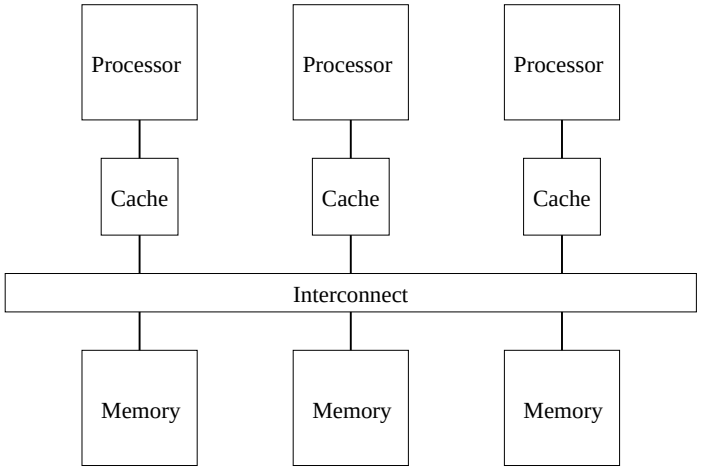
When we speak of a *process* in this paper, we mean a schedulable entity. Our *process* could be a *thread* in many systems.

We often refer to *gaining* or *holding* a *lock*. The *lock*, in this case, is the mechanism by which we guarantee mutually exclusive access to a resource or critical section. The *critical section* comprises the part of the process that uses the protected *resource*. We use the concepts of *holding a lock or resource* and *occupying a critical section* interchangeably.

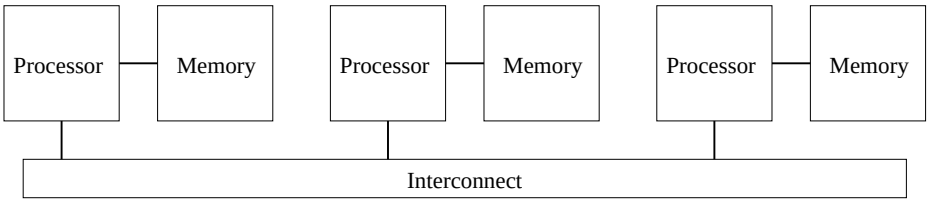
The data structures we use generally include three record types, *Locks*, *Processes*, and *Requests*. There is one Lock record for each lock that can be requested. With queue locks, the Lock record might or might not actually contain an indication of whether the resource is locked; our Lock records contain only pointers. There is one Process record per process. To request a lock, a process places a Request record (and sometimes its own Process record) on the queue of requests for the lock. In all of the schemes, a lock being released is explicitly *granted* to fill one of the requests on the list. The remaining requests are still *pending*.

The results reported here assume a loosely defined shared-memory multiprocessor architecture. The machine is byte-addressed, with 32-bit virtual addresses and a 32-bit word size.

We use one type of read-modify-write instruction, an atomic swap (*fetch-and-store*) of 32-bit values between a register and a location in memory. During a swap to a shared location, no other access to the same location interleaves with the parts of the swap.



(a) Coherent Cache Machine



(b) NUMA Machine

Figure 1: Memory Models

We use two simple memory models. Our generic coherent-cache machine is shown in Figure 1(a). Each processor is connected through its own cache to a shared interconnect and then to the shared memory. Any private physical memory that a processor might have (not shown) is used only for non-shared locations. The access time is uniform for a processor that reads any shared location that's not in its cache. The caches maintain coherence of the shared memory. In our model, a process that repeatedly reads a location without any intervening writes to the same cache block by other processes only uses the interconnect once, on the first read, to get the location into its cache. A process that repeatedly writes and reads a location without any intervening reads or writes of the same cache block by other processes only uses the interconnect once, on the first write, to invalidate the block in all other caches. The location we are swapping is not generally already in our cache, so the atomic swap generally involves two uses of the interconnect, one to read the block and one to invalidate it.

Our generic NUMA machine is shown in Figure 1(b). Each processor is connected to a

local memory and to a shared interconnect that connects it to other processors. Any cache that a processor might have (not shown) is used only for non-shared locations. The shared memory of the machine is spread across the local memories of the processors. The times for a processor to read or write shared locations is highly non-uniform. The read or write of a shared location in a processor's local memory (*local access*) is "much faster" than that of a location in another processor's shared memory (*remote access*). In addition, the remote access uses the interconnect while the local access does not. Again, our swaps are generally to remote memory, so we assume that an atomic swap makes twice the use of the interconnect that a simple read or write does. Due to the non-uniformity of memory access, it's important to force certain data structures to be allocated in a processor's local memory. We assume that processes and data structures do not migrate between processors on the NUMA machine.

3 Queuing Spin Locks on Machines with Coherent Caches

Starting with Graunke and Thakkar's FIFO-queuing spin lock [GT90], we must solve two problems to adapt it to a priority queue. First, a process that has just granted the lock to an arbitrary waiter must be able to know when its Request record is available for it to request the lock again. We cannot depend on the FIFO nature of the queue as Graunke and Thakkar do. Second, the lock holder must be able to find the highest-priority waiter.

Solving the first problem leads to our version of the FIFO lock, described in Section 3.1 and Appendix A. The difficulty is that a process cannot know when the Request it has granted is no longer being watched by another process and can be reused. A process does know, however, that the Request it has watched is available as soon as it's granted.

We describe our FIFO lock before proceeding to the problem of finding the highest-priority waiter.

3.1 FIFO Queue with Coherent Cache

We explain our FIFO lock for machines with coherent caches (*coherent-FIFO* lock) in this section by showing pseudo-code of the type definitions and initialization code and a diagram of the data structure operation. The whole pseudo-code is shown in Appendix A.

For the coherent-FIFO lock, the record definitions and the routines to initialize the structure are shown in Figure 2. Figure 3(a) labels our graphic representations for the Process, Request, and Lock records of this lock.

The routine *new_block_aligned* allocates a Request record and assigns its address to a pointer

```
type Lock = record
                tail : ^Request  // Request to be watched by next
            end record           //   requester

type Process = record
                   watch : ^Request  // Request that grants lock to me
                   myreq : ^Request  // Request that I grant when thru
               end record

type Request = record
                   state : (PENDING, GRANTED)
               end record

procedure init_Lock (var L : Lock)

    new_block_aligned (L.tail)  // allocate a Request for the Lock
    L.tail^.state := GRANTED    // mark the Request as GRANTED (lock
                                //   is free)
end procedure

procedure init_Process (var P : Process)

    new_block_aligned (P.myreq)  // allocate a Request for the Process

end procedure
```

Figure 2: Record Definitions and Initialization Code for Coherent-FIFO Lock

variable, just as *new* does in Pascal, but *new_block_aligned* guarantees that the Request starts on a new cache-block boundary and occupies one or more whole blocks. We thus prevent Requests from sharing cache blocks. That, in turn, prevents two processes from spinning on the same cache block.

The routine *request_lock* is called to acquire a lock. In that routine, a process (say P) marks a Request as PENDING and provides it to the lock for P's successor to spin on (*watch*) in exchange for the Request left there by P's predecessor. P then spins on the predecessor's request record until it's granted, at which time P holds the lock and can proceed to use the protected resource.

Figure 3(b) shows a system after initialization with one lock and three processes, where P_1 has already set its Request to PENDING and is about to enqueue the Request. Figure 3(c) shows the situation just after P_1's Request is enqueued and P_1 holds the lock. The request by P_1 is followed in order by those of P_2 and P_3, respectively. All three requests are shown in the queue in Figure 4(a). At that point, P_1 still holds the lock and P_2 and P_3 are spinning in their caches, adding no load to the interconnect.

The routine *grant_lock* is called to release a lock. In that routine, the lock holder simply grants

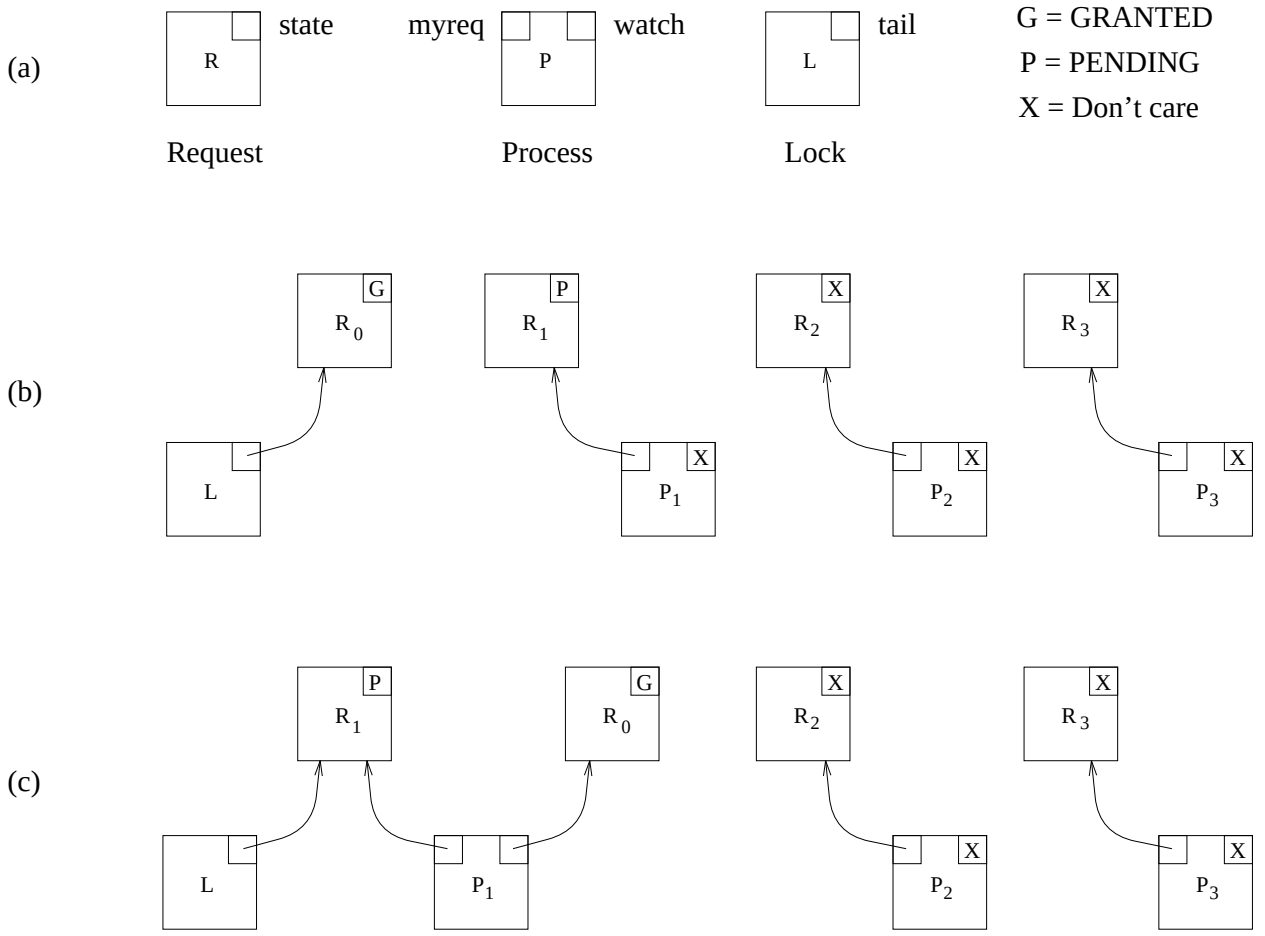


Figure 3: Coherent-FIFO Lock Structure in Operation

the Request (*myreq*) that it put on the queue in the first place, allowing the next requester to obtain the lock. Then the releaser alters its own Process record to take ownership of the Request that was granted to it by its predecessor.

Figure 4(b) shows the structure after P_1 has passed the lock to P_2 and taken ownership of R_0. Finally, Figure 4(c) shows the structure after P_2 and P_3 have both released the lock and it is unlocked.

A key idea in our algorithms, then, is to exchange ownership of Request records each time a process is granted the lock. When a lock is initialized, it is allocated a Request record that is marked as GRANTED. When a process is initialized, it is allocated a Request record, too. A side effect of this change is to remove the requirement for a Request record per lock per process in the Graunke and Thakkar scheme ($O(L * P)$ Requests in a system with L locks and P processes). Our scheme uses just one Request per lock or process in the system ($O(L + P)$ Requests). Besides saving space, it seems easier to manage our structures in a system where the number of locks and/or processes might not be known beforehand.

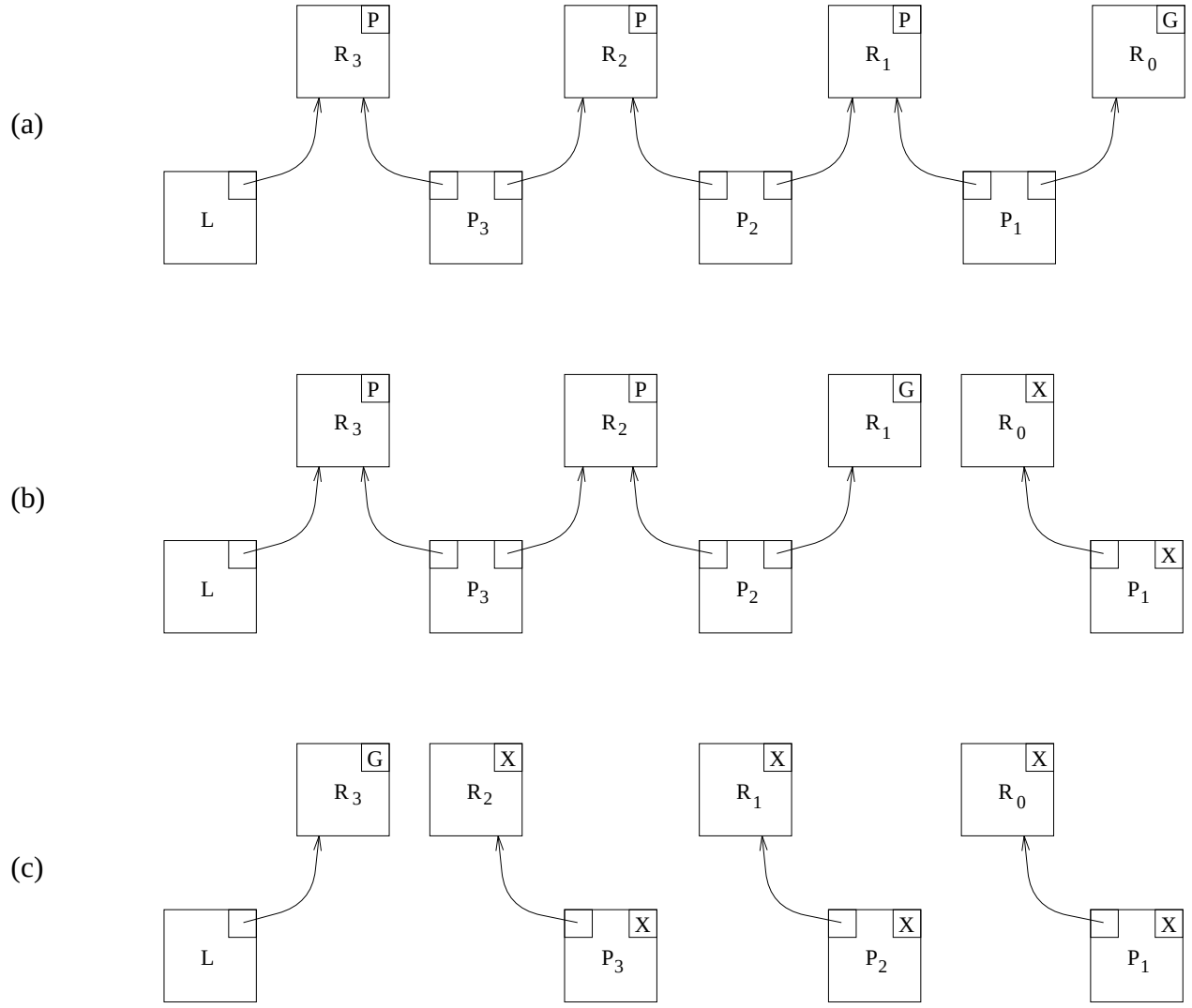


Figure 4: Coherent-FIFO Lock Structure in Operation (cont'd.)

3.2 Priority Queue with Coherent Cache

Note, in Figure 4(a), that our queue of waiters differs from a conventional linked data structure in that it cannot be passively traversed. While conventional queues contain only passive (data) elements and may be traversed by a single active process, our queue consists of alternating passive and active elements. The only way that activity can proceed along the queue is by the sequential granting of requests by the processes in the queue.

To make a priority queue lock for a coherent-cache machine (*coherent-priority* lock), we add pointers that allow the lock holder to traverse the list of waiters from the oldest request to the newest one on the list. The new pointers include a head pointer from the Lock record

to the oldest Request and a pointer from a Request to the Process that is watching it. We also add a back pointer from a Request to the Process that owns it (placed it on the queue), to simplify writing the algorithm. Thus, a Process and the Request that it owns are doubly linked to each other.

The pseudo-code for our coherent-priority lock is shown in Appendix B and its operation is diagrammed in Figure 5. For this and later algorithms, we leave out the initialization code. The graphical representation is labeled in Figure 5(a) and a newly initialized lock and process are shown in Figure 5(b).

When a process enqueues its request (Figure 5(c)), it receives a pointer to the Request that it should watch, just as in the FIFO case. It then stores a backpointer to its own Process record in the *watcher* field of the Request. The list structure is *stable* (pointers not being written by other than the lock holder) from the head Request down to the Request owned by the newest process that has set its backpointer. Figure 5(d) shows a lock with four processes on its queue. Process P_1 holds the lock and the list is stable all the way from R_0 to R_3, where P_4 has yet to set the *watcher* pointer.

The stable portion is the part of the list on which the lock holder can operate safely. The stable portion always includes the current lock holder. Thus, the lock holder can remove itself and its new Request (the one it watched) from the list, which is its first step in releasing the lock. In Figure 5(e), P_1 has removed itself and its new Request from the list.

The next step is to find the highest-priority waiter, which is done simply by traversing the stable portion of the list. As noted by Markatos and LeBlanc [ML91], we must first choose the highest-priority waiter and then grant it the lock. There is no atomic action to do both. Therefore, it's possible that, after the lock holder has traversed the stable portion of the list and chosen the highest priority waiter, but before it has granted the lock, more processes might finish enqueuing themselves. In that case, they are considered the next time the lock is released. By starting at the head of the list, we give the newcomers the best chance to finish enqueuing themselves before we reach them. Note, however, that any processes with requests after P_4's, even if fully enqueued, will not be seen at least until P_4 has finished enqueuing itself.

Finally, the lock holder simply grants the request of the highest-priority waiter in the same manner as it would grant the request of the oldest waiter in the FIFO queue. In the situation pictured in Figure 5(e), P_1 would choose P_3 as the highest-priority waiter and pass it the lock by setting the state of R_2 to GRANTED.

Figure 5(f) shows one more type of operation on the data structure. As shown there, P_3 is preparing to release the lock and has removed itself and its new Request record from the list. Their pointers are shown as dashed arcs to distinguish them from the arcs that form the remainder of the list. Also shown is that P_4 has finally finished enqueuing itself and will be considered in P_3's search for the highest-priority waiter.

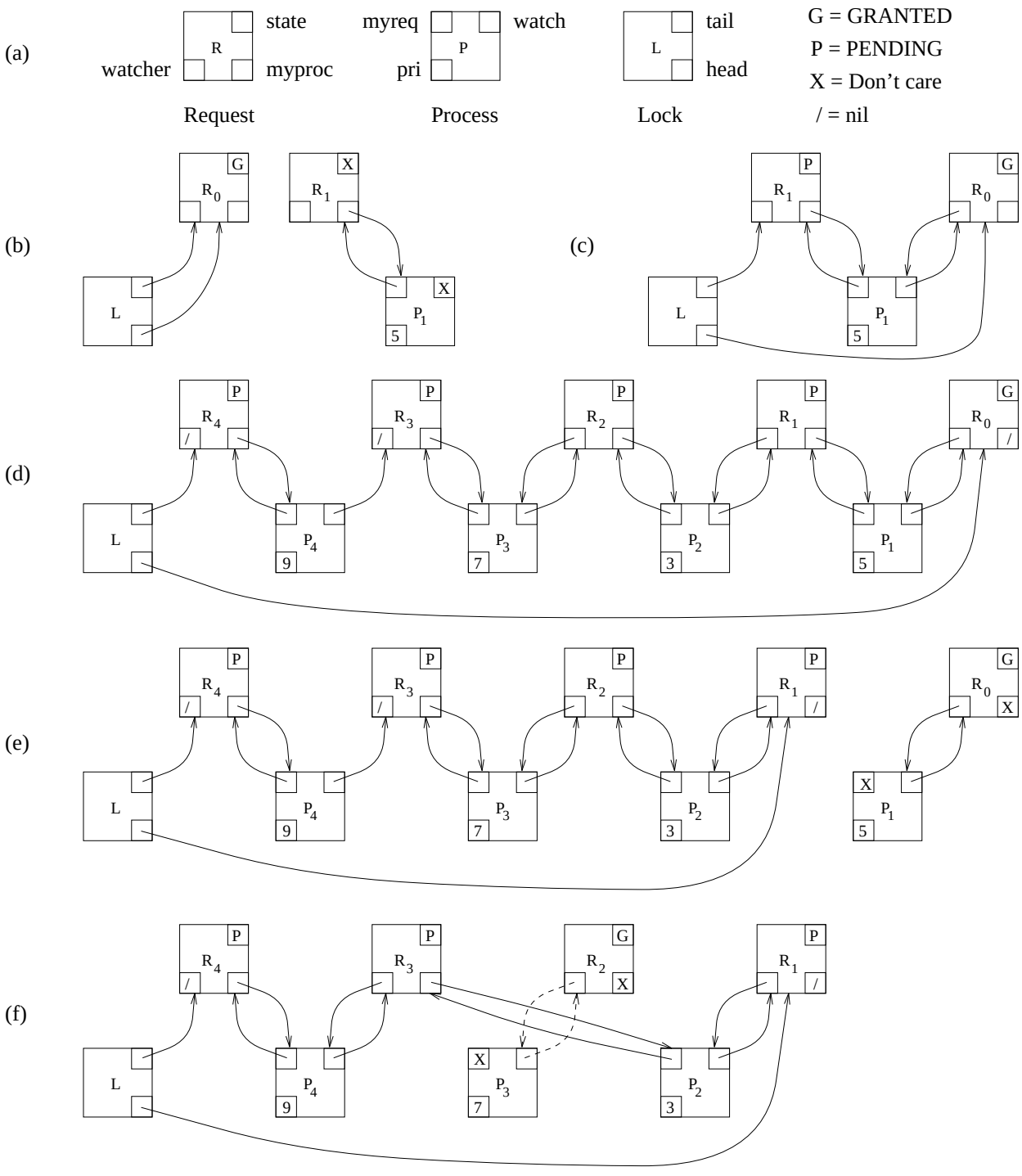


Figure 5: Coherent-Priority Lock Structure in Operation

4 Extending Our Scheme to a NUMA Machine

The problem in a NUMA machine is not merely to have different processors watch different locations. Without a cache coherence mechanism, each processor must spin on a location in its local shared memory. With our scheme of exchanging ownership of Requests, however, the physical location of the Request record that a process watches is unrelated to which processor is doing the watching. To adapt our scheme to the NUMA environment, then, we set up a pointer from the Request record back to the Process that's watching it. The waiter can then check the Request once and, if not GRANTED, spin on a location in its own (locally allocated) Process record. The granter, in turn, must mark the Request as GRANTED and then check for a pointer to a waiting Process. If it finds a pointer, it follows the pointer and sets the state of the waiting process to GRANTED.

To prevent a possible race between the waiter and the granter, we use a protocol of swaps in which either the GRANTED indication is passed to the waiter or the Process pointer is passed to the granter. To implement this scheme, we require Process pointers to be even numbers, so we can use the low-order bit of the Process pointer as the state of the Request. That way, the *watcher* and *state* fields of the Request record are combined in one 32-bit word and can be swapped as a unit. If the waiter performs its swap before the granter, then the waiter receives a state of PENDING and the granter receives the Process pointer. Otherwise, the waiter receives a state of GRANTED and the granter receives a nil pointer. The two possibilities are illustrated in Figure 6.

4.1 FIFO Queue on a NUMA Machine

The pseudo-code for our NUMA-FIFO lock is shown in Appendix C. The only other feature that requires explanation is the initial swap of the nil pointer into the *watcher* field and the PENDING status into the *state* field of the Request. Having very imprecisely defined the shared memory system for this machine, we use this swap to signify that we must guarantee that the nil pointer and the PENDING status are safely stored in the Request record before the pointer to that Request is placed in the Lock record in the next swap.

4.2 Priority Queue on a NUMA Machine

The pseudo-code for our NUMA-priority lock is shown in Appendix D. It simply combines the features of the coherent-priority lock and the NUMA-FIFO lock.

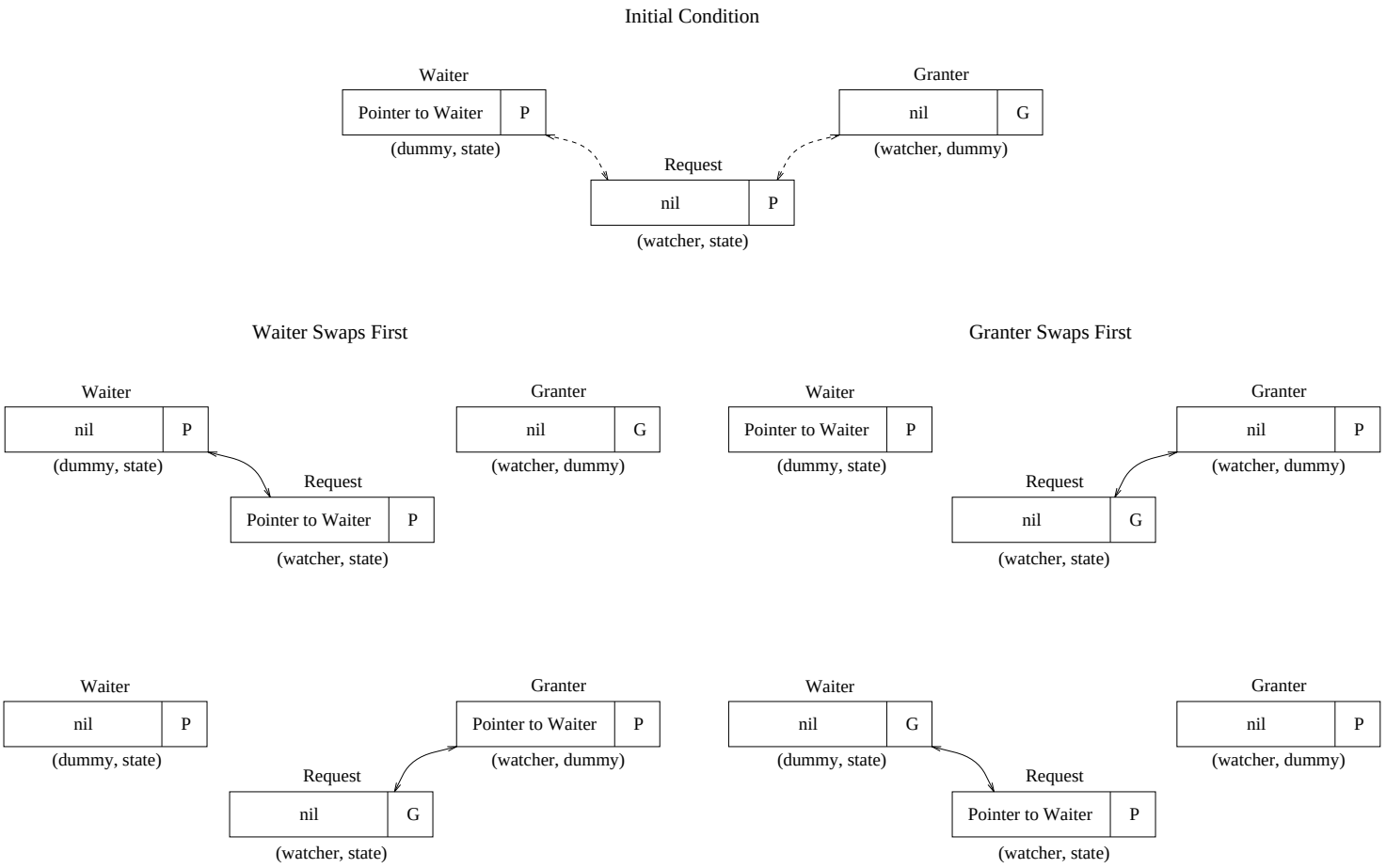


Figure 6: Operation of the Swap-Based Lock-Granting Protocol

5 Other Features

5.1 Speed of Selecting a Grantee

Markatos and LeBlanc [ML91] assume in their model that process priorities are dynamic, as they are in some real-time scheduling schemes (eg. earliest deadline first (EDF), least slack time). If priorities can change at any time, then it's necessary for each lock holder to consider all waiters when choosing a successor. In that case, it takes $O(n)$ time to grant the lock (where n is the number of waiters).

With static priorities (eg. rate monotonic scheduling), we can improve on that time complexity by maintaining an auxiliary data structure of waiters. For instance, a heap provides an average $O(log\ n)$ time to grant a lock. The idea here is that each lock holder starts with the L.head pointer and traverses the list of waiters, inserts them into the heap, and changes L.head to keep track of where it left off in the list. It then removes the highest-priority waiter from the heap and grants the lock to it.

Our scheme would also work with quasi-static priorities, where the relative priorities of existing tasks (a task is a particular execution of a process) are static. EDF is an example. In that case, the priorities of the waiters already in the heap are not changing relative to each other and new waiters can be added to the correct position in the heap.

We recognize that the added complication, time, and space of using the heap would not pay off unless the number of waiters were quite high, but use of the heap can be switched on and off adaptively, for any given lock, by the releasing routine.

5.2 Nested Locks

With the other schemes that use $O(L+P)$ space [MCS91, ML91], a lock-holding process that needs to acquire another (nested) lock must provide an additional Request record. Therefore, with nesting they require $O(L + P * D)$ space, where D is the depth of the nesting. We can, however, adapt our scheme to allow nested lock acquisitions without requiring any additional Request records.

For clarity, the algorithms in this paper have the lock holder take ownership of its new Request as part of the lock releasing procedure. To provide nested requests, though, a process must take ownership of its new Request as soon as the lock is granted to it (in the *request_lock* routine) and keep track of which locks to release or requests to grant next as it releases locks. (It releases locks in the reverse of the order that it acquired them.) Our routines to facilitate nested acquisition of FIFO locks (not presented here) form a linked stack of Requests to grant. Thus, the *myreq* field of the Process record still points to the

next Request to grant, but that Request is the top of the linked stack. Each Request contains a pointer to the next Request on the stack.

For a priority lock, we need a stack of Locks, not Requests, but we could use a very similar scheme. We would keep a linked stack of locks to release, with a "top" pointer in the Process record. Again, we would move the Request ownership transition code from *grant_lock* to *request_lock*. We could also dispense with the Lock argument that is passed to *grant_lock*.

5.3 Timeout

While not directly applicable to some real-time preemptive priority scheduling models, a timeout feature is of interest in the general context of locks. By a timeout feature, we mean that the lock request routine either obtains the lock before a time limit or returns a status indicating a failure to do so.

With a non-queuing spin lock, a waiter can simply include a time check in its spin loop and, after the timeout, stop spinning on the lock and return the "failed" status. With our queue lock scheme, however, we have the additional problems of keeping the request from being granted and removing the timed-out Request from the queue.

For a waiter to rescind its request for a queue lock, then, we must allow the waiter to somehow manipulate the queue. If the waiter simply stops spinning, its Request will eventually be granted without any process watching it. No process will ever watch that Request, so all other requests for the lock will hang up. We must ensure that the lock holder is assured of having granted the lock to a process that will use it.

Remember that the only safe operations on our queue structures are the enqueue operation performed by any process or processes, and the dequeue operation performed only by the lock holder. A waiter may, however, safely write to the Request that it's watching.

When timing out, the waiter (say P_1) must leave behind both an indication that it no longer desires the lock and a pointer to the next request. In setting up those indications, P_1 is racing with a potential granter. If the lock is granted to P_1 just before it marks its request as timed out, then P_1 is holding the lock and must either return a "success" status or pass the lock to the next waiter and return a "failed" status. Either policy is supported by our scheme.

Even a test-and-set would enable us to provide a simple timeout feature. With a test-and-set, the waiter would first set a pointer from the Request it's watching to the next one in line and then race for the Request by testing and setting the GRANTED flag. The granter would also test and set the flag. If the waiter finds the flag was already set, then it has lost the race and has received the lock. If the granter finds the flag was already set, then it must follow the pointer to find the next request to grant.

With the swap instruction available to us, we can combine the Request state with the pointer and use the swap to arbitrate the winner of the race, as we did with the *watcher* pointer in the NUMA machine. In fact, for the NUMA case, we can potentially combine the state with a pointer to either a Process or the next Request, by keeping the bottom two bits as zero in any legitimate pointer. The bottom bit is then the state of the Request and the second bit is an indication of whether the pointer in the remaining 30 bits points to a Process or to a Request. The keys to using the swap this way are that we happen to be able to pack enough information into 32 bits and that there are only two processes involved.

A major problem with this timeout scheme is that the process that's leaving the queue cannot take its Request record with it immediately, because the potential granter must use the information in the Request first. Thus, we must allow the granter to mark the Request as released (as by setting the pointer to nil) after it uses it.

When a process attempts to acquire a lock, then, it must first check whether its Request has been released and, if not, either spin until it is released or be able to supply a different Request record to the lock. If the process is attempting to acquire the same lock as before, it can swap a "pending" indication back into the Request record in a attempt to resume its previous position in the queue. If it succeeds, fine. Otherwise, it knows that the Request will be released very soon, if not already, and it can spin until the release and then reuse the Request.

With the possibility of a process needing to supply extra Request records, we lose our guarantee of $O(L + P)$ space and cannot bound the space requirement below $O(L * P)$ for locks that use this timeout scheme. Note, however, that the Request will be reached eventually in a FIFO queue and it will be reached on the next release or two in a priority queue that is completely scanned on each release.

5.4 Conditional Lock

A problem noted by Graunke and Thakkar [GT90] is that of allowing a conditional request for a lock, one that acquires the lock if it's immediately available and fails otherwise. We observe that a conditional request is very much like one with a zero timeout. After finding that the request it's watching has not been granted, the requester immediately executes the timeout action. Graunke has also developed a method to provide a conditional lock and remove the failed request from the queue [Gra92].

5.5 Preemption

As discussed by Markatos and LeBlanc [ML91], another lock-related issue is that of preemption. Clearly, if the holder of a lock is descheduled, use of the critical section stops until the

holder is scheduled again. With non-queuing locks, preempting a waiter doesn't block the use of the critical section, because the lock is not actively granted; it is actively acquired. Once preempted, the waiter is no longer waiting and is not at risk to acquire the lock.

With queuing locks, however, preemption of a process does not prevent that process from being granted the lock, which would turn it into a preempted lock holder. As noted in [ML91], however, we might want to allow schedulers to preempt waiters. To do so, we need a way for the scheduler to determine that a process is spinning for a lock and to deactivate the lock's request for the duration of the preemption. Our discussion of timeout suggests a way to rescind a request. Notice that preemption is similar to a timeout in that the goal is to cause a request not to be granted.

The scheduler must detect that a process it is preempting is waiting for a lock (and which lock) and do the timeout action for it. When rescheduling the process onto a CPU, it can attempt to reactivate the request. (If it hasn't been removed, the reactivation attempt succeeds.) If reactivation fails, the scheduler can queue a new request. Alternatively, it can start the process back at the beginning of the lock_request routine.

While it seems that a test-and-set works fine to timeout a request, it might not allow reactivation of a request. Swap, however, allows the requester to swap PENDING back for the TIMEDOUT state and know whether it gets back TIMEDOUT or GRANTED.

In any case, we must still work out the details of detecting that a process is waiting for a lock and backing it out of that condition. Recent work with non-queuing locks [BRE92] offers some promising techniques.

6 Discussion

6.1 Testing

We have implemented nesting versions of all four of our algorithms and performed the minimal functional testing that we could do in a non-multiprogrammed environment (simple mutual exclusion, lock acquisition and lock release). For the FIFO locks, we also tested nested lock acquisition and release. For the NUMA locks, we tested an older version of the protocol that handles the race between a waiter setting a pointer to itself and the lock holder granting its request. That protocol didn't involve a swap, but would not be reliable under some reasonable assumptions about read/write ordering in a NUMA machine.

We have also implemented a nesting version of our coherent-FIFO lock on the Proteus multiprocessor simulator [BD91]. One benefit of working on the simulator is that we can monitor the functional performance of the lock scheme. Our prototype includes a detector

for violations of mutual exclusion that has been triggered only when we purposely altered the lock acquisition routine to test the detector. We have also observed that locks are granted only in FIFO order.

So far, we have simulated between 1 and 18 processes (each on its own processor) contending for a single lock. We are still working on generating quantitative output from the simulation, but we've made two observations that provide some optimism. First, the cache hit rate is much less sensitive to contention with our queuing lock than it is with a simple test-and-set lock. Second, we get a very high hit rate when we have only one process repeatedly entering and leaving the critical section.

6.2 Characterizing Spin Lock Schemes

[MCS91] nicely identifies four general traits by which to characterize and compare spin lock algorithms. We generalize these traits and add a fifth one of our own (atomic instruction set) to create the following list of traits and our comments on them. The first two traits are environmental, involving architectural realities that might not be controllable by the system designer. The importance of the third trait depends on the application at hand. The last two are performance-oriented results of the type of lock used.

- Target machine memory system (coherent-cache or NUMA). A scheme should be optimized for the type of memory system on which it must run, but we've found that designing an algorithm for a NUMA machine was more challenging than doing one for a coherent-cache machine. With a coherent-cache machine, there is one extra active element, the coherence mechanism, to ensure that the state of a Request is moved to a location that's local to the processor that's reading it. On the NUMA machine, we had to develop our swapping protocol (a special-purpose software-level coherence mechanism) to allow the releasing and requesting processes to cooperate in making the state of a request available locally for any spinning. In fact, any of the schemes that we've seen for NUMA machines should work reasonably well on coherent-cache machines, but not vice versa.

- Target machine atomic instruction set (compare-and-swap (C&S), swap, or test-and-set). Each of the locking schemes that we've mentioned or described requires a specific type of atomic instruction(s). Often only one (or none) of these three instructions is implemented on a particular machine. We haven't done a serious survey of available architectures, but we believe that the most generally applicable lock would use just test-and-set. Given a machine with an atomic swap instruction (eg. multiprocessors that use the swap available in the Intel 80x86 [Int90] or Motorola 88x00 [Mot91]), we also believe that our algorithms offer better characteristics than would a queuing spin lock that used the swap as a test-and-set. The algorithms that use compare-and-swap

offer some potential performance advantages over ours in NUMA machines, but they also require a swap instruction. Neither swap nor compare-and-swap may be trivially emulated by the other. The actual performance difference would depend on the relative expense of the swap and compare-and-swap instructions on a given machine. Finally, we have not yet looked seriously at our work in the context of other fetch-and-op instructions, load-linked/store-conditional [KH92], or transactional memory [HM92].

- Order in which the lock is granted to waiters (random, FIFO, or priority). For general parallel programming, FIFO order might be marginally better than random order, because it's fair and prevents starvation. We haven't yet determined the specific applicability of priority-ordering to priority-scheduled real-time systems, but we believe that it offers potential improvements in schedulability. Note also that being able to grant a lock in priority order automatically allows us to grant it in nearly FIFO order, by assigning time stamps to requests (depending on the availability, synchronization, and precision of a real-time clock). A scheme that provides FIFO order, however, does not generally provide priority order.

- Load imposed on the interconnect (remote spinning vs. no remote spinning). Much of our effort has been dedicated to minimizing this load. Our key goal here, which we (and most others) have accomplished, is to eliminate any spinning on remote locations. Next, we would like to minimize the number of atomic instructions executed over the interconnect. Finally, remote reads and writes should be minimized.

- Space required per lock ($L * P$, $L + P * D$, or $L + P$). The simplest spin lock typically requires just a byte per lock. All of the queuing spin lock schemes require more than that and our goal is generally to minimize the space required. Ours is the only scheme to provide the lowest asymptotic space ($O(L + P)$) with nesting. The exact space taken, however, depends on the sizes of the individual records. The key advantage of our scheme is that we can statically allocate one Request record per process when creating the process and not worry about how many to allocate for each process.

Table 1 presents a number of queuing spin lock schemes in terms of these characteristics. For coherent-cache machines, it includes the locks of Graunke and Thakkar [GT90], Markatos and LeBlanc [ML91], and ours. For NUMA machines, it includes the locks of Mellor-Crummey and Scott (with and without compare-and-swap) [MCS91], Markatos and LeBlanc, and ours.

7 Summary and Future Work

Our main technical contributions are our techniques and algorithms that provide tight control over lock grant order, use only the atomic swap instruction, use at most one (local only) spin for lock acquisition and no spinning for lock release, and need only $O(L + P)$ space on either

Category	Trait	[GT90]	[ML91] Coherent	Our Coherent	[MCS91] (C&S)	[MCS91] (Swap only)	[ML91] NUMA	Our NUMA
Memory Model	Coh. Cache	X	X	X	*	*	*	*
	NUMA				X	X	X	X
Atomic Instruction Set	C&S				X		X	
	Swap	X		X	X	X	X	X
	Test&Set		X					
Grant Order	FIFO	X	* linear	X	X	usually	*	X
	Priority		X	X			X	X
Interconnect Load	Remote spin						rarely	
	No remote spin	X	X	X	X	X	usually	X
Space	$L * P$	X	X					
	$L + P * D$				w/nest	w/nest	w/nest	
	$L + P$			X	w/o nest	w/o nest	w/o nest	X

Table 1: Queuing Spin Lock Schemes Compared by Five Traits

In the table, an "X" indicates a primary trait of a scheme, while an "*" indicates a trait that is covered somewhat by default. The "linear" waiting in the case of Markatos and LeBlanc's coherent-cache lock is due to its derivation from Burns's lock [Bur78]. Linear waiting is not first-come-first-served but says that when a process (say P) is waiting, no other process will get the lock more than once before P gets the lock.

a coherent-cache or NUMA machine. An additional contribution is our outline of techniques to extend the basic algorithms.

The details of even the basic algorithms that are presented here have remained in a state of flux, so we expect to make at least some minor changes to them, particularly when applying them to specific machines, to optimize their performance.

We also need to:

- Complete our investigation of the extensions, producing working models of each idea for testing.
- Prove the correctness of the final algorithms.
- Implement our remaining algorithms on the simulator to experiment with their functionality and performance.
- Implement the other published algorithms on the simulator, for comparison.
- Test our algorithms and the others on actual multiprocessors, to calibrate the simulator and to obtain more data on performance.
- Complete some analysis we've begun to compare the interconnect loads of the algorithms using our architectural models and extend it to specific machines.
- Conduct a more complete survey of available architectures, to judge the specific applicability of our algorithms.

- Consider how our techniques fit into the context of other types of atomic instructions and how they relate to lock-free synchronization technology.

If our algorithms perform well, then we plan to use our techniques in several additional lines of research in the general and real-time systems area:

- Quantify the contribution of our algorithms to improving the schedulability results for priority-scheduled real-time systems.
- Investigate whether our locks are useful tools to use in non-priority-scheduled real-time systems.
- Develop abstract data types to better encapsulate our data and control structures.
- Work on supporting higher-level general and real-time synchronization primitives with our low-level ones.

References

[And90] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[BD91] Eric A. Brewer and Chrysthanos N. Dellarocas. *PROTEUS User Documentation, Version 0.2.* MIT, Cambridge, Massachusetts, 1991.

[BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, October 1992.

[Bur78] J. E. Burns. Mutual Exclusion with Linear Waiting using Binary Shared Variables. *SIGACT News*, 10(2), Summer 1978.

[Gra92] Gary Graunke. Personal communication, December 1992.

[GT90] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[HM92] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. Technical Report CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, December 1992.

[Int90] Intel Corporation, Santa Clara, California. *Microprocessors*, 1990.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[ML91] Evangelos P. Markatos and Thomas J. LeBlanc. Multiprocessor Synchronization Primitives with Priorities. Technical report, University of Rochester, Rochester, NY, 1991.

[Mot91] Motorola, Inc., Phoenix, Arizona. *MC88110: Second Generation RISC Microprocessor User's Manual*, 1991.

[MSZ90] Lory D. Molesky, Chia Shen, and Goran Zlokapa. Predictable Synchronization Mechanisms for Real-Time Systems. *Real-Time Systems*, 2(3):163–180, September 1990.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

A Coherent-Cache FIFO Queue Lock Algorithm

```
type Lock = record
                tail : ^Request  // Request to be watched by next
            end record           //   requester

type Process = record
                   watch : ^Request  // Request that grants lock to me
                   myreq : ^Request  // Request that I grant when thru
               end record

type Request = record
                   state : (PENDING, GRANTED)
               end record

procedure request_lock (var L : Lock, var P : Process)

    P.myreq^.state := PENDING                // tell successor to wait
    P.watch := fetch&store (L.tail, P.myreq) // enqueue my request

    loop until (P.watch^.state = GRANTED)    // wait until predecessor
                                             //   finishes
end procedure

procedure grant_lock (var P : Process)

    P.myreq^.state := GRANTED  // pass lock to successor
    P.myreq := P.watch  // take ownership of the Request that I
                        //   watched (to use it for my next request)
end procedure

procedure init_Lock (var L : Lock)

    new_block_aligned (L.tail)  // allocate a Request for the Lock
    L.tail^.state := GRANTED    // mark the Request as GRANTED (lock
                                //   is free)
end procedure

procedure init_Process (var P : Process)

    new_block_aligned (P.myreq)  // allocate a Request for the Process

end procedure

atomic function fetch&store (var X : Word, newvalue : Word) : Word

    fetch&store := X  // the two operations on X in this function occur
    X := newvalue     //   without any intervening reads or writes of X

end function
```

B Coherent-Cache Priority Queue Lock Algorithm

```
type Lock = record
                tail : ^Request  // most recently enqueued Request
                head : ^Request  // least recently enqueued Request
            end record

type Process = record
                   pri : Priority
                   watch : ^Request
                   myreq : ^Request
               end record

type Request = record
                   state : (PENDING, GRANTED)
                   watcher : ^Process  // extra pointers to allow
                   myproc : ^Process   //   list traversal
               end record

procedure request_lock (var L : Lock, var P : Process)

    P.myreq^.state := PENDING
    P.myreq^.watcher := NIL
    P.watch := fetch&store (L.tail, P.myreq)
    P.watch^.watcher := addr(P)  // point the request I'm watching
                                 //   back to me
    loop until (P.watch^.state = GRANTED)

end procedure

procedure grant_lock (var L : Lock, var P : Process)

    var
        highpri : Priority
        highreq : ^Request
        currproc : ^Process

// remove my Process and the Request I watched from the list

    P.myreq^.myproc := P.watch^.myproc

    if (P.myreq^.myproc <> NIL) then
        P.myreq^.myproc^.myreq := P.myreq
    else
        L.head := P.myreq
    end if

// search the list for the highest-priority waiter

    highpri := LOWEST'PRIORITY - 1
    highreq := L.head
    currproc := L.head^.watcher

    while (currproc <> NIL) do
        if (currproc^.pri > highpri) then
```

```
            highpri := currproc^.pri
            highreq := currproc^.watch
        end if
        currproc := currproc^.myreq^.watcher
    end while

// pass the lock to the highest-priority waiter

    highreq^.state := GRANTED

// take ownership of the Request that I watched

    P.myreq := P.watch
    P.myreq^.myproc := addr(P)

end procedure
```

C NUMA Machine FIFO Queue Lock Algorithm

```
type Lock = record
                tail : ^Request
            end record

type Process = record
                   localstate : (PENDING, GRANTED)
                   watch : ^Request
                   myreq : ^Request
               end record

// Note:  The notation in the following declaration indicates
//        that the Request record has one word in it.  The
//        high-order 31 bits of the word are the high-order
//        31 bits of watcher, which is a pointer to a Process.
//        The low bit of the word is the state field, which can
//        hold one of two values, PENDING or GRANTED.  The low-
//        order bit of watcher is implicitly always 0, so we must
//        ensure that all Process pointers are even numbers.  We
//        combine the pointer and the state of the Request this
//        way so that we can swap them as a unit.

type Request = record
                   (watcher : ^Process, state : (PENDING, GRANTED))
               end record

procedure request_lock (var L : Lock, var P : Process)

var
    (dummy : ^Process, state : (PENDING, GRANTED))

    fetch&store ((P.myreq^.watcher, P.myreq^.state), (NIL, PENDING))
    P.localstate := PENDING

    P.watch := fetch&store (L.tail, P.myreq)

    (dummy, state) := fetch&store ((P.watch^.watcher, P.watch^.state),
                                                   (addr(P), PENDING))
    if (state = PENDING) then
        loop until (P.localstate = GRANTED)
    end if

end procedure

procedure grant_lock (var P : Process)

    var
        (watcher : ^Request, dummy : (PENDING, GRANTED))

    (watcher, dummy) := fetch&store (P.myreq^.watcher, P.myreq^.state),
                                                         (NIL, GRANTED))
    if (watcher <> NIL) then
        watcher^.localstate := GRANTED
    end if
```

```
    P.myreq := P.watch
end procedure
```

D NUMA Machine Priority Queue Lock Algorithm

```
type Lock = record
                tail : ^Request
                head : ^Request
            end record

type Process = record
                localstate : (PENDING, GRANTED)
                pri : Priority
                watch : ^Request
                myreq : ^Request
              end record

type Request = record
                myproc : ^Process
                (watcher : ^Process, state : (PENDING, GRANTED))
              end record

procedure request_lock (var L : Lock, var P : Process)

var
    (dummy : ^Process, state : (PENDING, GRANTED))

    fetch&store ((P.myreq^.watcher, P.myreq^.state), (NIL, PENDING))
    P.localstate := PENDING

    P.watch := fetch&store (L.tail, P.myreq)

    (dummy, state) := fetch&store ((P.watch^.watcher, P.watch^.state),
                                                    (addr(P), PENDING))
    if (state = PENDING) then
        loop until (P.localstate = GRANTED)
    end if

end procedure

procedure grant_lock (var L : Lock, var P : Process)

    var
        highpri : Priority
        highreq : ^Request
        currproc : ^Process
        (watcher : ^Request, dummy : (PENDING, GRANTED))

    P.myreq^.myproc := P.watch^.myproc

    if (P.myreq^.myproc <> NIL) then
        P.myreq^.myproc^.myreq := P.myreq
    else
        L.head := P.myreq
    end if

    highpri := LOWEST'PRIORITY - 1
    highreq := L.head
    currproc := L.head^.watcher
```

```
    while (currproc <> NIL) do  // while currproc is a pointer
        if (currproc^.pri > highpri) then
            highpri := currproc^.pri
            highreq := currproc^.watch
        end if
        currproc := currproc^.myreq^.watcher
    end while

    (watcher, dummy) := fetch&store ((highreq^.watcher, highreq^.state),
                                                        (NIL, GRANTED))
    if (watcher <> NIL) then
        watcher^.localstate := GRANTED
    end if

    P.myreq := P.watch
    P.myreq^.myproc := addr(P)

end procedure
```